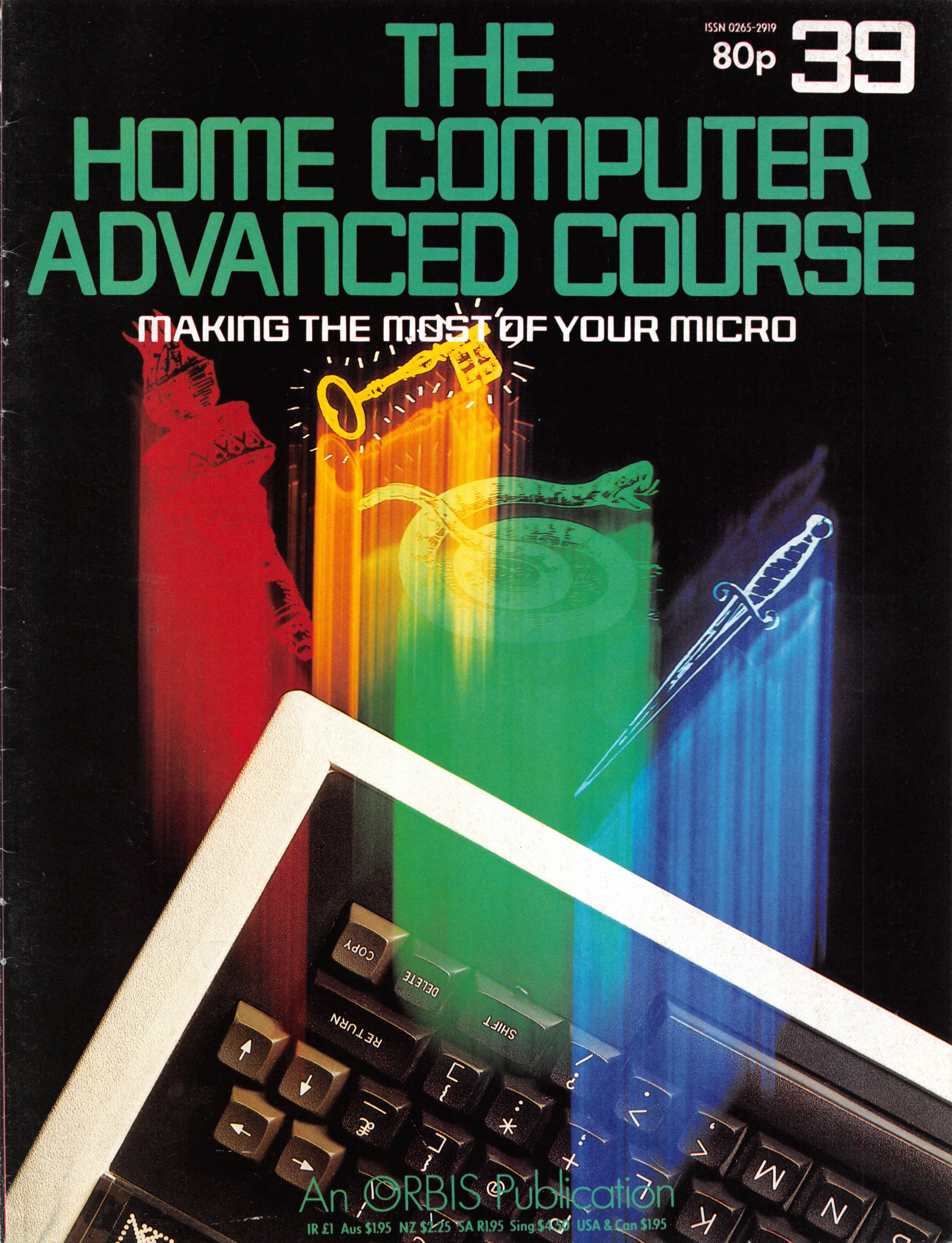
ISSN 0265-2919

80p 39

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **We look at the new Commodore 16, the long-awaited replacement for the Vic-20.**
- **Continuing our series of programming projects, we examine how to develop adventure games in BASIC and LOGO.**
- **Lotus 1-2-3 has already been examined in our series on integrated software, now we take a closer look at its spreadsheet capabilities.**

QUIZ

1) If the pivot point is five centimetres from a servo motor, what is the maximum weight that can be lifted by the servo?
2) By what number would you have to AND a character cell on a Spectrum in order to 'mask' the PAPER colour?
3) What is the major difficulty in the digital storage of speech?
4) Today, holograms are mostly known for producing three-dimensional pictures. For which purpose are they potentially useful in computers?

Answers To Last Week's Quiz

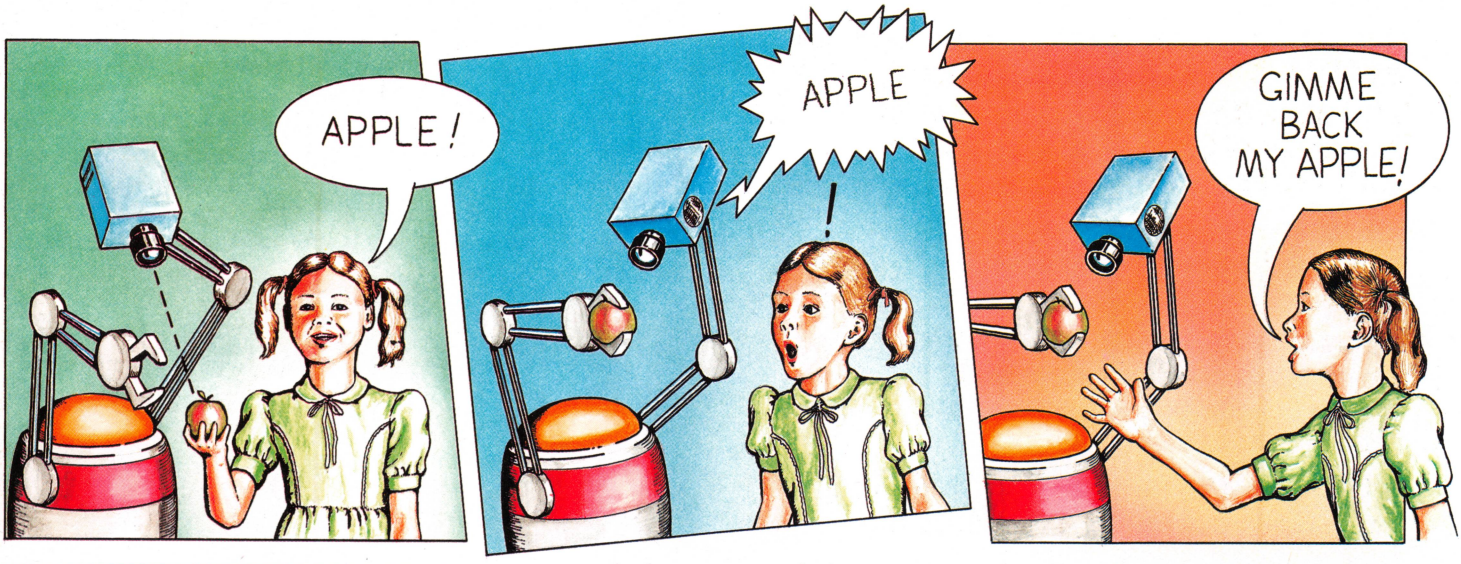
1) Flyerfox uses the linear predictive coding method of speech synthesis.
2) To run Graph Plan a Z80 second processor is required.
3) The 'grey scale' is the range of shades of grey (usually 256) that can be discerned by a computer.
4) By increasing the number of digital pulses, we can create a higher resolution and so more accurately emulate the sine wave.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Steve Malone; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY PAUL CHAVE

ON SPEAKING TERMS



STEVE CROSS

Seeing Is Believing
When a human sees an object, like an apple, and applies a name to it, there is an understanding of the meaning of 'apple'. The robot can visually recognise the object by matching what it sees with an internal image, and can repeat the sound pattern it has stored to go with the apple. But the robot has no understanding that the object is an edible fruit, nor, perhaps more importantly, that the apple actually 'belongs' to the human. This, of course, is something the human understands perfectly

Speech is one of the most difficult tasks for a robot to achieve and the reason for this is because the way in which humans learn to speak is not fully understood. In order to understand some of the problems associated with robot speech, therefore, it is necessary to discuss some of the important theories of language acquisition.

The study of human speech has produced two schools of thought: those who believe that language skills are innate — something that we are born with — and those who believe that language is acquired, or learnt. Those psychologists who argue that language is innate point out that man is the only creature to communicate by language. Those who believe it is acquired cite experiments with animals that have been taught to communicate successfully with humans by sign language.

If people learn speech simply by being exposed to it then it would make sense to look for a method of making robots do the same. After all, it would make life so much easier if a robot could learn the language just by listening to you speak it.

Certain limited attempts have been made to enable a computer to expand its knowledge of grammar by being given extra examples of grammatical sentence structures, while other experiments have tried to allow a robot to learn new words and morphemes (language elements) in any language simply by being shown them. But no system has yet been devised that has succeeded in teaching a robot to learn speech.

So, for all practical purposes, robot language skills are dependent on the assumption that language is innate, that the skills are not learned, and what we must do is to work out the rules of language and embed them permanently into the robot as if the robot had been born with them. In general, this consists of two distinct phases: syntactic analysis and semantic analysis.

Syntactic analysis is concerned with the grammar of what is being said and decodes the surface structure of the message or encodes the message into a grammatical form ready for transmission by the robot. The most common method of doing this is by means of a 'parsing tree' that gradually breaks down, or builds up, a sentence from the various parts of speech. It isn't an easy task — but it is a task that is gradually being tackled with some success.

Semantic analysis is much harder and involves working out the sense of the message (when the robot is listening to you speak); or working out what message needs to be conveyed (when it wants to speak to you). The problem with semantic analysis is that language is not context-free — its meaning depends upon the context in which it is spoken (and this does not apply to the spoken context alone, but to the entire context of the message). This context may encompass knowledge about the state of the world as one speaks, as well as the knowledge that each party has of the other.

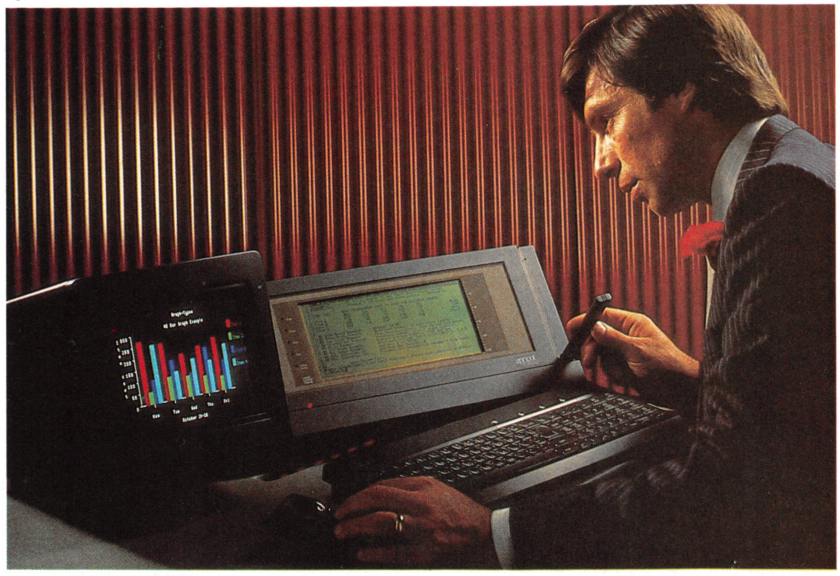
This approach has been adopted in experiments conducted by the computer scientist Terry Winograd, who wrote a program that enabled a robot to understand what was said to it and to act on instructions. However, Winograd used a computer simulation of a robot that was only able to operate in a very closely-defined world. In this case, its world consisted of a number of building blocks that it was able to manipulate. Winograd's program, known as SHRDLU, was able to make a

good semantic analysis, but the world it was able to make sense of was extremely simple. A robot working in the chaos of the real world would have had much greater difficulty understanding what was being said to it.

In order for robots to employ language usefully, there must be a message that is passed between you and the robot and vice versa. It is relatively easy for robots to speak, because anything that a robot would be likely to want to express would be very limited, since its knowledge is so restricted. It is much more difficult for a robot to understand what you might want to say to it, because anything that you might wish to communicate is much more difficult to analyse.

At one time it was thought that speech input to robots would be analysed by carrying out a syntactic analysis of the input and that this would reveal the meaning of the message. But recent work has shown the importance of knowledge of the surrounding world and the context in which the message is spoken. This has led to experiments in which a tentative syntactic analysis of the speech signal is made in order to make a first guess at the meaning. Then, in the light of what the robot knows about the world and the likely things that might be said in its world, the robot revises its original syntactic analysis in the hope of gradually homing in on a correct analysis of what is being said. However, this is far beyond what any commercially available robot can currently do. Here, we will look at how contemporary robot systems speak and understand speech.

Apricot F1



Hearing And Speaking
Robot and computer speech is fairly simple to create. Speech synthesis devices, like the Currah shown here, are available for even the smallest home computers. But recognition is more difficult because of the variations in the way humans pronounce vowel sounds, and because of the amount of processing power and memory required to handle a vocabulary of more than a few words. Systems like Big Ears, and the Apricot F1, have a small range of recognisable commands built in, but too few to cover more than a minimal number of operations

SPEECH SYNTHESIS

The simplest method of speech synthesis employs a tape recorder in which a message spoken by a human being is recorded on tape and played back by the robot at an appropriate time. This might not seem to be quite what you had in mind when you first thought of robot speech — but it is the starting point of all speech synthesis systems. We will look at the limitations of this method and then see how we can improve on them.

The most obvious limitation is that a tape recorder is mechanical, expensive, bulky and liable to break down. So the next step is to take the same message and convert it into digital form so that it can be stored on a chip in the robot's memory. This is done using an analogue-to-digital converter, in which numbers are used to represent the continuously varying waveform of speech. This is exactly the same method that is used in the digital recording of music on, for instance, compact disc systems.

This method has its drawbacks, too. One of the main problems is that a digitised signal takes up a lot of room in memory. Compact disc recording samples the acoustic signal around 44,000 times per second with a resolution of approximately 16 bits (i.e. the amplitude of the waveform at any moment is stored as a 16-bit number, which enables 2^{16} levels to be discerned, where 2^{16} = 65,536). Using this system, each second of the recording would occupy 88,000 bytes of memory. Clearly, a spoken message exceeds the storage capacity of any microcomputer. However, this sampling rate is only applicable to high fidelity sound reproduction; a simple speech system could be operated with a resolution of eight bits, and a sampling rate of 3,000 samples per second, which only uses up three Kbytes of memory!

Big Ears

COURTESY OF WILLIAM STUART SYSTEMS LTD

Currah

However, in order to free the maximum amount of memory space, further economies need to be made. Linguists have found that spoken language can be conveniently broken up into units of speech called phonemes. In all, there are generally agreed to be some 40 different phonemes for most spoken languages, so it is possible to store the exact acoustic information necessary to describe each of these 40 phonemes and then use these as the foundation of robot speech. Typically, the phoneme information is held on a commercially manufactured speech synthesiser chip and all the

robot has to do is to string together those phonemes to generate the required message. This message is usually held as a string of phoneme numbers in the computer's memory.

Most of the speech synthesisers in use can be programmed by writing out the message that the robot is to speak in a phonetic version of English. Thus, the message 'Can you come here?' might be written as 'kan yew kum heah' and this would be sufficient for the synthesiser chip to produce the correct string of sounds. This is not exactly the same notation that linguists use when describing phonemes — they have their own specialised alphabet — but it suffices for robots.

At this point you will notice that the robot is no longer using a pre-recorded message — it is actually generating messages of its own. Because of this it is possible to make the robot say anything we wish without the need for having the whole message stored beforehand.

So, if we wanted to we could try programming in some of the rules of grammar in an attempt to make the robot say quite original things. But, as already mentioned, the number of different things that a robot might want to say is fairly limited so there is no need for too much complexity unless we happen to be feeling either adventurous, or curious to see what can be done.

If you have ever heard a speech synthesiser on a robot you will know that the quality of the speech, although usually comprehensible, is by no means perfect. This is due to two factors. The first is that the form that a phoneme takes when used by a human speaker varies considerably depending on the phonemes that precede and follow it. The second is that the overall sound of human speech varies depending on the meaning that we wish to convey. 'Will you sit down?' and 'Will you sit *down*?' are two identical written messages, but they will sound quite different if the first is said by a courteous host to his guest and the second is uttered by an exasperated schoolteacher. Some attempts have been made to capture this intonation in speech synthesis systems, but it is difficult to apply as a robot has no knowledge of the meaning of the words it is speaking.

SPEECH RECOGNITION

The inherent problem to solve when devising a speech recognition system is that the things we may wish to say to a robot, and the different ways in which we might express them, are many and varied. The problem could be approached by using a tape recording of everything that we might want the robot to understand. When we spoke, it could then simply scan through all of its tape recordings and look for the one most like the message it just heard — and that, in principle, is how many robots do recognise speech. They store internal 'templates' of spoken messages and, on being spoken to, simply look for the template that offers the best match. These templates are usually obtained by training the robot — repeating a word or phrase several times — until it has an 'average' template of what we have said. This method works well if you only have a small number of things to say to the robot and are going to say them in roughly the same way every time. It is used for robots that respond to simple commands such as 'forward', 'turn left', and so on.

However, this is a comparatively simple problem and is known as 'discrete speech recognition' because each spoken item is 'discrete' — that is to say, it is separated from other messages by a slight pause during which nothing is said.

The real problem emerges when we wish to speak to the robot using 'continuous speech', which is the type of speech we normally use when speaking to each other. Try saying 'It's a nice summer day' and listen closely to what you said. You will find that it comes out as something like 'Itssan ice ummerday' with the words and sounds running into each other.

The way that people tackle this problem when they are listening to others speaking is by guessing what it is that the speaker means to say — not usually a hard task — and using this guessing to decode the message. But for a robot to do this it would have to know a great deal about what was likely to be said and what it was likely to mean — a very complex task.

In general, speech synthesis by robots is becoming quite common, although there is still room for improvement in the quality of their speech. Speech recognition is a much more difficult task and, currently, the best that can easily be achieved is to endow the robot with an understanding of speech equivalent to a well-trained dog that responds to spoken commands, as long as there are not too many of them. However, there is a tremendous interest in solving all of the problems of robot speech and the next few years are likely to see substantial advances.

Hear Me, Feel Me
The Voicemate is a voice-controlled robot arm developed for laboratory and industrial use by the science engineering department at Newcastle Polytechnic

COURTESY OF NEWCASTLE POLYTECHNIC

GRID IRON

Continuing our investigation of spreadsheet modelling, we now examine some of the many advanced features found in Multiplan, a comprehensive electronic worksheet from Microsoft for the Commodore 64.

Microsoft incorporated many advanced ideas, developed from earlier spreadsheet packages, into its first spreadsheet program, Multiplan. These include the ability to act on groups of cells that are described by name; to sort a group of entries according to a specified criterion; to split screens, allowing different areas of the spreadsheet to be viewed simultaneously; to search quickly through information held in table form and then output a requested value; and to perform IF...THEN conditional constructs, among others. Originally available only on high-level computers such as the IBM PC and Apple II, Multiplan has recently become available on the Commodore 64.

The model we will build with Multiplan simplifies the task of keeping track of statistics. The data we use relates to American football, but the structure can be adapted to other sports.

Once Multiplan is loaded, a standard format worksheet of 63 columns and 255 rows is displayed. Rows and columns are both numbered, so the home cell, in the upper left-hand corner, is referred to as cell R1C1 — for Row 1 Column 1. A menu of command options appears at the bottom of the screen, with a cursor highlighting the first choice, Alpha. Multiplan menu commands can be chosen by pressing the Space bar to move the cursor to the desired command and pressing Return, or by typing the first letter of the command.

Selecting a command often causes a sub-menu to be displayed, offering a wide variety of options for formatting data, memory management, and so on. Pressing a letter key accesses a command, so you have to type A for Alpha before entering text. Numbers can be entered directly, but formulae must be preceded by a plus (+) or equals (=) sign.

The first two rows of the worksheet hold titles. For convenience, we have formatted the cells from R1C1 to R2C5 for continuous text, which allows text to extend beyond cell boundaries. This is accomplished by typing:

```
F(ormat) C(ells) R1C1:R2C5
```

then placing the cursor over the word Cont and pressing Return. The colon is used to indicate a range of cells. Some columns have been widened or shortened to accommodate their entries.

There are two main portions of the worksheet: one holds information for a specific team over a nine-week period, and the other is a table of the won/lost records for all the teams in the same 'conference' (see the American Football box for an explanation of terms). Once the skeleton of the model has been constructed, much of the weekly data will have to be entered by hand, with just a few formulae to keep running totals as the season goes on.

League Records

Team Performance Record

IAN McKINNELL

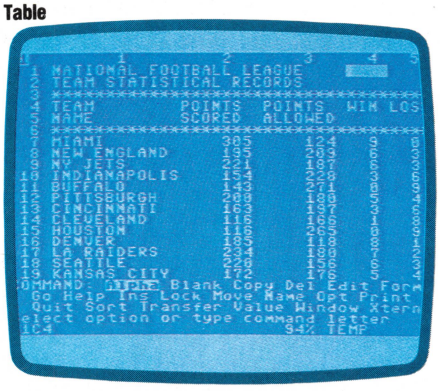
The first portion of the worksheet is a table. The totals in the table must be updated each week, after the teams have played. By storing the information in a table, we can take advantage of one of Multiplan's advanced features: the SORT facility. We have entered the team names, categories and data as shown. The initial ordering of the teams is based on current standings in the league. However, the table can be sorted by any of the categories stored. Multiplan will sort a specified range of rows in a given column, in ascending or descending numerical order. Text is, of course, sorted alphabetically.

As an example of how the SORT function works, we will rearrange the data shown by team name in alphabetical order. After typing S for SORT, Multiplan displays the following:

```
SORT by column:_ between rows:_ and:_
order:> <
```

We want to sort by Column 1 between rows 7 and 21, in ascending (>) order. Pressing Return causes Multiplan to reorganise the names alphabetically, and rearrange the data to match. For example, all the numbers attached to Miami in our original list move with Miami to its new position. Simply by changing the key column in our SORT command, we can also rearrange the table according to the highest scoring team, the teams that have allowed their opponents to score the fewest points, etc.

Sorted Table



Now scroll the screen by pressing the down cursor arrow and find the second portion of the worksheet — the individual team performance record. Two formulae will be used here. The first is a simple SUM formula, to keep a running total of the weekly values. Find the column labelled TOTALS in section two (R24C12). We will want Multiplan to add up the values in each weekly column. Since we will want to copy the formula so that totals are found for all our categories — covering the area from R25C12 through R32C12 — we need to incorporate a relative cell reference. In Multiplan, this is done simply by pointing at the active cells with the cursor.

The formula is entered by typing:

```
=SUM(
```

and then pressing the left arrow key until the cursor rests on R25C3. We then type a colon to indicate that a range of cells is being specified. The cursor automatically returns to the cell in which you are entering the formula, so press the left arrow once, with the cursor resting in R25C11, and then press Return. The formula should now look like this:

```
=SUM(R[-9]C:R[-1]C)
```

and totals for the values held in the described range should be displayed. Now copy the formula into the range of cells from R26C12 through R32C12 by keeping the cursor on the formula and using the Copy command:

```
C(opy) d(own) 7 rows
```

Use the same process to find the totals for YDS RUSH and YDS PASS. The SUM formula is placed in cell R27C3, for yards gained on offense, and R31C3, for yards yielded to the opposing team. The formula is copied to the right eight columns, to cover the full nine-week period.

The second formula, using the IF statement, is a little more complicated, but extremely useful. In our model, we will let Multiplan determine whether a game has been won or lost by comparing the point totals in two categories: Points Scored (by our team) and Points Allowed (the opponents' score). We need a statement like this: If Points Scored > Points Allowed, print WIN, else print LOSS.

Once again we want relative references, and so we use cursor movements to point out the locations of the two values. Place the cursor in R34C3, labelled WIN/LOSS, and enter the formula:

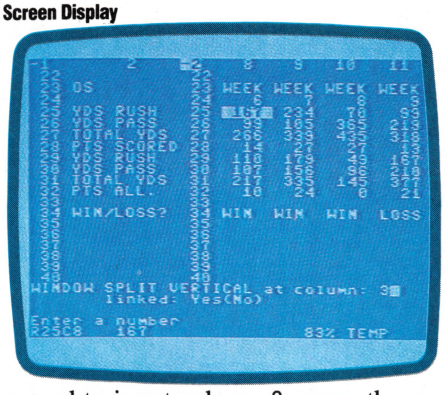
```
IF(R[—6]C>R[—2]C,"WIN","LOSS")
```

Conditional (IF...THEN) Construct

Note that text used within formulae must be enclosed in double quotation marks, and parentheses are required surrounding the conditions. Now copy the formula across the row, as before. The model we have created contains data for nine weeks. Because of the screen size, we cannot see either the labels on the left edge of our team record, or the totals on the right. But we can split the screen into two windows, which can be scrolled together or independently.

We will want to split the screen vertically at column 3, so we press W(indow) and S(plit), followed by V(ertical). Multiplan will then display: WINDOW SPLIT VERTICAL at column:_linked YES/NO.

Split Screen Display



You need to input column 3, move the cursor to NO, and press Return. If the windows are not linked, they can be scrolled separately. Now the labels can be seen no matter what portion of the worksheet is being viewed. To close the window, you type W(indow) C(lose), followed by its number.

In the next instalment of the course, we will be looking at one of the more refined offerings among spreadsheets — Lotus 1-2-3.

KEVIN JONES

American Football

For those unfamiliar with American football, here is a brief explanation of the terms used. Two teams of 11 players take turns trying to move a cigar-shaped ball across a goal line. Opposing goals are 100 yards apart. The ball can be carried by a runner, thrown forward as a pass, or kicked between the goalposts. Three points are scored for a kick (called a field goal); six points are awarded for a carry or pass across the line (called a touchdown); and one for a kick following a touchdown (appropriately called a point after touchdown).

Each team has four attempts, or 'downs', to move the ball 10 yards closer to the goal. If successful, they can continue toward the goal with another four downs. When a player carries the ball, it is called a rush, and the number of yards 'rushed' by a team is an indication of how far the ball has been carried during the game. The number of yards 'passing' means how far the ball has been thrown.

There are two 'conferences' in the National Football League (the USA's primary professional league) — the American Conference and the National Conference. Teams play a 16-game season that begins in September and culminates in the Super Bowl in January. The Super Bowl is contested by the best team from each conference

SPIRIT OF ADVENTURE

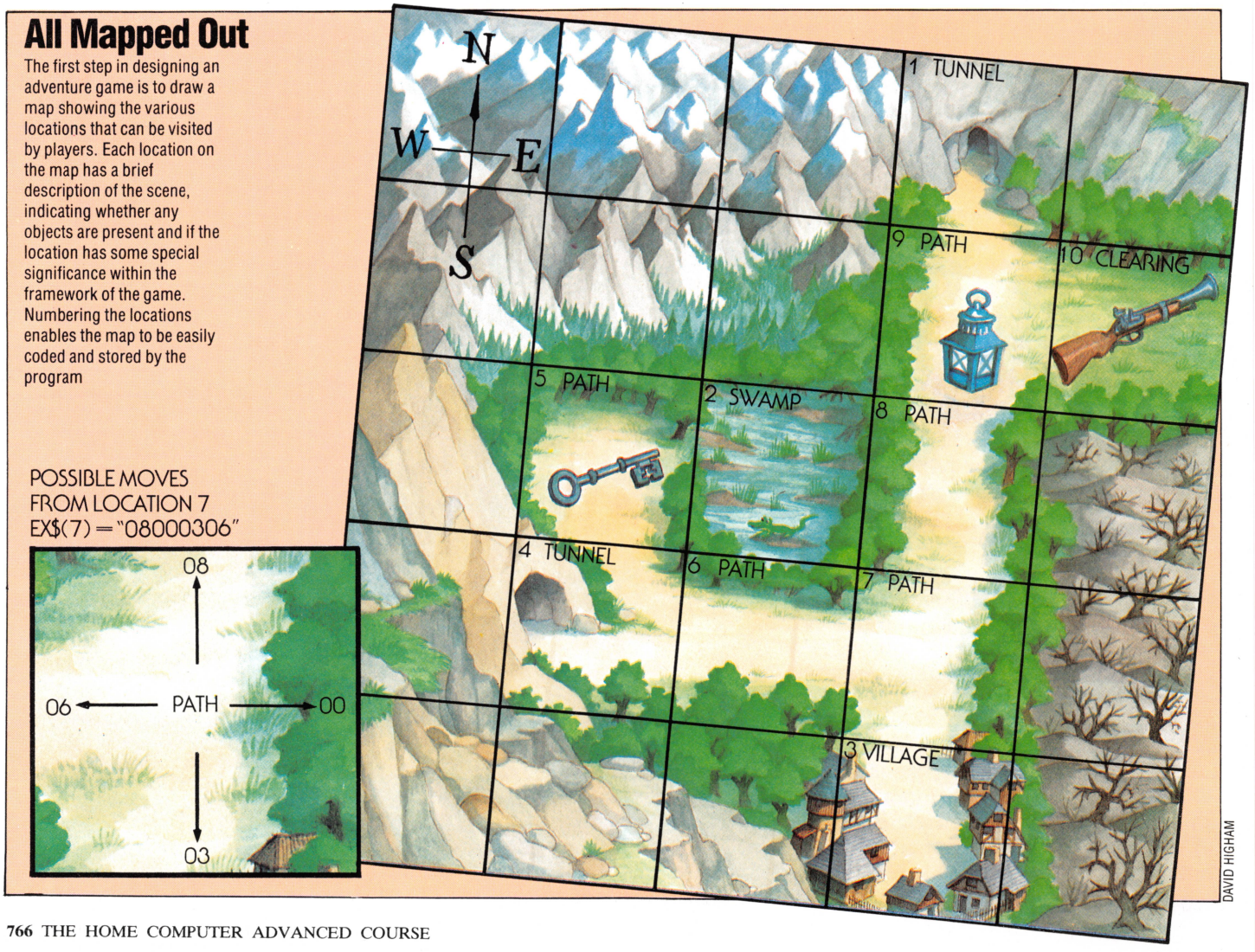
Adventure gaming is an extremely popular pastime among home computer users. But playing a game is only half the fun; writing your own adventure is an enjoyable and creative activity. We begin an extensive programming project in which we take you through all the stages of building up an adventure game.

Adventure game playing became popular in the early 1970s when the game Dungeons and Dragons was devised. In this game, the players take on the roles of various characters within an imaginary world designed by the Dungeonmaster. This imagined world generally consists of an intricate maze of rooms, containing objects and perils, which the players have to negotiate. Generally, the aim of the game is to escape from the maze, usually rescuing someone or something along the way. Mainframe programmers were the first to apply the game to computers, constructing complex labyrinths for other mainframe users to wander through. The advantage of computer-based Dungeons and Dragons was that the Dungeonmaster and the players did not have to be present at the same time, allowing individuals to play whenever they wished. Since then the Dungeons and Dragons type of game has widened its scope and appeal considerably — the Multi-User Dungeon (see page 384) is an excellent example of how sophisticated some have become.

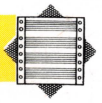
Some adventure games are purely text-based, whereas others make use of colour and graphics to provide screen images. However, some critics argue that the addition of graphics uses up valuable memory space that could otherwise be used to add intricacies to the game's structure. They also point out that a computer graphic picture of a scene or location is no match for one's

All Mapped Out

The first step in designing an adventure game is to draw a map showing the various locations that can be visited by players. Each location on the map has a brief description of the scene, indicating whether any objects are present and if the location has some special significance within the framework of the game. Numbering the locations enables the map to be easily coded and stored by the program



DAVID HIGHAM



own imagination conjuring up a picture based on a textual description. However, the rise in the popularity of adventure games is almost certainly attributable to the enhanced visual appeal that graphics give, and, although some recent micro games use only simple pictures to enhance the text, others attempt to make the game visual.

In our programming project we shall be looking at the techniques involved in programming an adventure game. During the project, you will be given sections of a listing to an adventure game called Digitaya, which will build into a complete program. In this game, the player is cast as an 'electronic' agent given the task of descending into a microcomputer to locate and rescue the mysterious Digitaya from the clutches of the machine. There are many dangers and difficulties along the way and you have to use all your knowledge of computers to good effect to escape unharmed. The program is, as far as possible, written in 'standard' BASIC, with 'Flavours' given where appropriate. Therefore, provided you have sufficient memory capacity, the program will run on your computer. As we are going to discuss the various programming techniques in detail, it would be difficult not to give away many of the secrets of the game, and this would spoil, to some extent, the pleasure of playing it when it is complete. We will, therefore, construct a shorter game called Haunted Forest, in parallel with Digitaya, which will demonstrate the techniques and algorithms used to build the larger game.

MAKING A MAP

The starting point for the design of our adventure game is to construct a map of the fantasy world that we are imagining. On this map, we mark out the various locations within the world, the position of any objects to be found, and signify those locations that are considered 'special'. Most locations on the map will simply allow the player to move in and out of them, and pick up or drop any objects that are there. Special locations may be perilous (a swamp or a place where a dragon lurks), or they may require a series of special actions to be performed before you can enter into, or exit from, them.

The best way to begin making a map is to consider roughly how many locations are needed for the game. Haunted Forest has 10 locations and was designed on a five by five grid (as shown in the illustration), whereas Digitaya has nearly 60 locations and was originally designed using a 10 by 10 grid.

The grid squares are initially unnumbered and the designer starts by filling in locations on the map. On the Haunted Forest map there is a path, two tunnels, a swamp, a clearing and a village. The positions of several objects are also marked at the bottom of the squares where they are located. Those locations marked with an asterisk (*) are 'special' and will be treated in a different way to the rest of the locations.

Once the layout has been finalised we can number each location. The only special consideration we have taken into account in choosing the location number is that all the special locations have been numbered first. The order in which the others are numbered is not important, but once numbers have been selected it is important that they are not changed later.

PROGRAMMING THE MAP DATA

The first programming task is to convert the information in the map into data for the program. There are many ways of doing this, but what we will do here is use two one-dimensional arrays to hold the map data. The first array, LN$(), holds descriptions of each location. For example, for location 7, LN$(7) will contain 'on a path'. When the data is used later in the program to describe a location it will be prefixed by the words 'You are'.

The second array, EX$(), holds data about the possible moves that can be made out of a location. Both of our games limit themselves to four directions: North, East, South and West. EX$() provides information about the location number to be moved to for each of the four directions. The data is held as a string made up of eight digits. The location number for each direction is entered in the order NESW, using a two-digit number for each direction.

For example, location 7 has exits to the North, South and West, but none to the East. The first two digits of EX$(7) are 08 (not just 8), which shows that location 8 is to the North. The second pair of digits, 00, indicates that there is no exit in this direction (East). The digit pairs 03 and 06 represent the locations found to the South and West of location 7. Using this system, up to 39 locations could be specified; if more than this were required then the data for EX$() would have to be entered as groups of three digits.

The three objects in the Haunted Forest are read into another array — IV$(,). This two-dimensional array keeps track of the position of each object as it is moved around the forest. Each object has a description and its starting location on the map. For example, IV$(C,1) may be GUN, and its position at the start of the game is given by IV$(C,2), As the objects are carried around during the game the position members of the array will be updated accordingly.

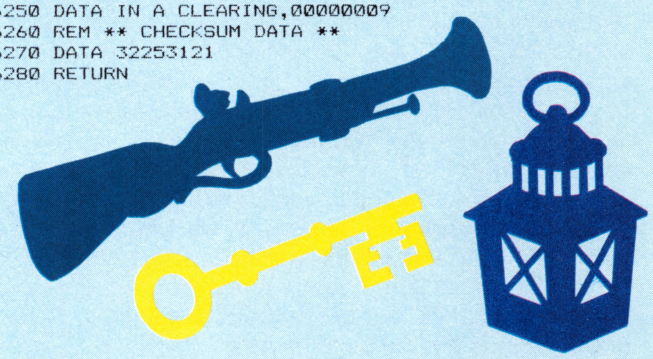
At the end of the map data in both of our listings there is another item of data. This is a 'checksum' and is given to ensure that the direction data has been typed in correctly. This is done by calculating a running total of the data values, which is compared against the checksum. If these are not the same then a mistake has been made and the program will stop running. You will notice that in Digitaya two checksums are used. This is because the total sum of all the direction data is too large to be held easily in one checksum, so a total for the left-hand and right-hand four digits is calculated separately. In the next instalment of the project, we will design routines to handle and display the map data assembled here.

Digitaya

```
6090 REM **** READ ARRAY DATA S/R ****
6100 REM ** READ INVENTORY **
6110 DIM IV$(8,2),IC$(4)
6120 FOR C =1TO8
6130 READ IV$(C,1),IV$(C,2)
6140 NEXT C
6150 :
6160 REM ** READ LOCATION & EXIT DATA **
6170 DIM LN$(55),EX$(55)
6180 C1=0:C2=0:REM INITIALISE CHECKSUMS
6190 FOR C=1TO54
6200 READ LN$(C),EX$(C)
6210 C1=C1+VAL(LEFT$(EX$(C),4))
6220 C2=C2+VAL(RIGHT$(EX$(C),4))
6230 NEXT C
6240 READ CA:IFCA<>C1THEN PRINT"CHECKSUM ERROR":ST
OP
6250 READ CB:IFCB<>C2THEN PRINT"CHECKSUM ERROR":ST
OP
6260 RETURN
6270 REM **** INVENTORY DATA ****
6280 DATA ADDRESS NUMBER,45,KEY,34,LASER SHIELD,25
6290 DATA TICKET TO THE TRI-STATE,26,DATA CREDIT C
ARD,28
6300 DATA DIGITAYA,30,CODE BOOK,19,BUFFER ACTIVATI
NG DEVICE,13
6310 :
6320 REM **** LOCATION & EXIT DATA ****
6330 DATA IN THE TV OUTLET,00000000
6340 DATA IN THE USER PORT,00090100
6350 DATA IN THE CASSETTE PORT,00110000
6360 DATA IN THE JOYSTICK PORT,00130000
6370 DATA IN A TRI-STATE DEVICE,00170000
6380 DATA IN THE ARITHMETIC & LOGIC UNIT,00310016
6390 DATA AT THE GATEWAY TO MEMORY,00490000
6400 DATA ON THE I/O HIGHWAY,09000001
6410 DATA ON THE I/O HIGHWAY,10000802
6420 DATA ON THE I/O HIGHWAY,11000900
6430 DATA ON THE I/O HIGHWAY,12001003
6440 DATA ON THE I/O HIGHWAY,13531100
6450 DATA ON THE I/O HIGHWAY,14001204
6460 DATA ON THE I/O HIGHWAY,15001300
6470 DATA ON THE I/O HIGHWAY A SIGN SAYS 'S OUT H'
,00001400
6480 DATA IN THE DATA REGISTER,00061700
6490 DATA ON AN 8 LANE HIGHWAY,16001805
6500 DATA ON AN 8 LANE HIGHWAY,17001900
6510 DATA ON AN 8 LANE HIGHWAY,18002000
6520 DATA ON AN 8 LANE HIGHWAY,19292100
6530 DATA ON AN 8 LANE HIGHWAY,20282200
6540 DATA ON AN 8 LANE HIGHWAY,21272300
6550 DATA ON AN 8 LANE HIGHWAY,22262400
6560 DATA ON AN 8 LANE HIGHWAY,23250000
6570 DATA IN THE CHARACTER MATRIX,26360024
6580 DATA HIGH IN THE MEMORY,27352523
6590 DATA IN THE MIDDLE OF MEMORY,28342622
6600 DATA IN THE MIDDLE OF MEMORY,29332721
6610 DATA LOW IN THE MEMORY,00542820
6620 DATA IN THE ACCUMULATOR'S LAIR,00000600
6630 DATA IN A LONG CORRIDOR,00420006
6640 DATA IN AN INDEX REGISTER,31000000
6650 DATA LOW IN THE MEMORY,54403428
6660 DATA IN THE MIDDLE OF MEMORY,33393527
6670 DATA HIGH UP IN MEMORY,34383626
6680 DATA IN THE CHARACTER MATRIX,35370025
6690 DATA IN A RANDOM VECTOR TABLE,00000000
6700 DATA HIGH IN MEMORY OVERLOOKING A HIGHWAY,390
03735
6710 DATA IN THE MIDDLE OF MEMORY,40003834
6720 DATA IN MEMORY - TO THE EAST IS A GATEWAY,410
03933
6730 DATA LOW IN MEMORY,00004054
6740 DATA IN A CORRIDOR,00430031
6750 DATA IN A CORRIDOR,00440042
6760 DATA IN A CORRIDOR,00004543
6770 DATA IN THE ADDRESS REGISTER,00004600
6780 DATA ON A 16 LANE HIGHWAY,45004700
6790 DATA ON A 16 LANE HIGHWAY,46004800
6800 DATA ON A 16 LANE HIGHWAY,47004900
6810 DATA ON A 16 LANE HIGHWAY A LARGE GATE LOOMS
TO THE WEST,48005007
6820 DATA ON A 16 LANE HIGHWAY,49005100
6830 DATA ON A 16 LANE HIGHWAY,50005200
6840 DATA ON A 16 LANE HIGHWAY,51000000
6850 DATA IN A VECTOR TO MEMORY,00290012
6860 DATA LOW IN MEMORY,00413329
6870 REM ** CHECKSUM DATA **
6880 DATA 100169,103973
```

Haunted Forest

```
6000 REM **** READ OBJ & MAP DATA ****
6010 DIM IV$(3,2),LN$(10),EX$(10),IC$(2)
6020 FOR C=1 TO 3
6030 READ IV$(C,1),IV$(C,2)
6040 NEXT C
6050 :
6060 FOR C=1 TO 10
6065 READ LN$(C),EX$(C)
6070 CC=CC+VAL(EX$(C)):REM CHECKSUM TOTAL
6080 NEXT C
6090 :
6100 READ CD:IFCD<>CC THENPRINT"CHECKSUM ERROR":STOP
6110 :
6120 REM ** OBJECT DATA **
6130 DATA GUN,10,LAMP,9,KEY,5
6140 :
6150 REM ** MAP DATA **
6160 DATA NEAR A TUNNEL ENTRANCE,00000900
6170 DATA IN A SWAMP,00000000
6180 DATA IN A VILLAGE,07000000
6190 DATA NEAR A TUNNEL ENTRANCE,05060000
6200 DATA ON A PATH,00020400
6210 DATA ON A PATH,02070004
6220 DATA ON A PATH,08000306
6230 DATA ON A PATH,09000702
6240 DATA ON A PATH,01100800
6250 DATA IN A CLEARING,00000009
6260 REM ** CHECKSUM DATA **
6270 DATA 32253121
6280 RETURN
```



Basic Flavours

Our listings were written for the Commodore 64. The following changes need to be made for these machines:

Spectrum:

For the Digitaya listing, make these alterations:

```
6110 DIM V$(8,2,24):DIM I$(4,24)
6170 DIM L$(55,52):DIM E$(55,8)
6200 READ L$(C),E$(C)
6210 LET C1=C1+VAL(E$(C)(TO 4))
6220 LET C2=C2+VAL(E$(C)(LEN(E$(C))-3 TO))
```

In addition, all the DATA values must be enclosed in quotes, except for the checksum data on line 6880.

For the Haunted Forest listing, make these alterations:

```
6010 DIM V$(3,2,5):DIM L$(10,22)
6015 DIM E$(10,8):DIM I$(2,5)
6030 READ V$(C,1)V$(C,2)
6065 READ L$(C),E$(C)
6067 LET CC=0
6070 LET CC=CC+VAL(E$(C))
```

In addition, all the DATA values must be enclosed in quotes, except for the checksum data on line 6270.

BBC Micro:

The following line must be added to the Haunted Forest listing:

```
6067 CC=0
```

No changes need be made to the Digitaya listing.

H

HOLLERITH CODE

Designed by Herman Hollerith (1860-1929) in 1888, the *Hollerith code* is a method by which letters of the alphabet, the decimal digits 0 to 9, and special characters can be coded onto a punched card. The card is divided into 80 columns and 12 rows. Each column represents a single character by holes that are punched in either one, two or three of the rows. The card is then read by a tabulator machine, or card reader, which processes the information.

Herman Hollerith
The inventor of the Hollerith Code for processing data on punch cards, Hollerith founded the tabulating machine company that became IBM

HOLOGRAPHIC MEMORY

Recently-issued credit cards and bank cards carry a small three-dimensional imprinted design known as a 'hologram'. This is produced as an interference pattern on photographic material, usually by a source of high-intensity radiation. Banks and credit card companies are hoping that the use of holograms will reduce the amount of credit card fraud by making it more difficult to forge cards. However, this particular application is insignificant compared with their enormous potential as a mass storage medium for computer data.

Holographic memory has been a reality in laboratories since the early 1970s. It involves coding binary information as an interference pattern on a photographic surface with a laser beam. Data can be read back from the hologram by projecting a low-power laser beam from behind. Holographic memory has the same advantages that laser discs have over other storage media — a holographic surface is highly resistant to environmental factors such as dust and extremes of temperature, as well as surface scratches. A holographic device created in the late 1970s could store 200 million bits of data on a 4in by 6in plastic card.

HOST COMPUTER

The terminal or computer that controls operations within a network is called the *host computer.* It can have many functions. In a microcomputer local area network (LAN), the host computer is primarily a 'server': it handles files, controls the flow of information and may act as a printer depository for the other nodes of the system. With more powerful systems, particularly in mainframe networks, the host computer may act as a switching device for time-sharing or multiple-user applications. Within a hierarchical communication system, with many levels of computer involvement, a host computer may act as a controller on one level, and at the same time serve as a working node on another level.

HOUSEKEEPING

Housekeeping tasks are program or operating system routines that keep computer operations running smoothly, without having a direct effect on the actual outcome of the program. The purpose of such routines is to keep things orderly and organised. Initialisation, garbage collection and memory management are typical housekeeping routines.

HUMAN FACTORS ENGINEERING

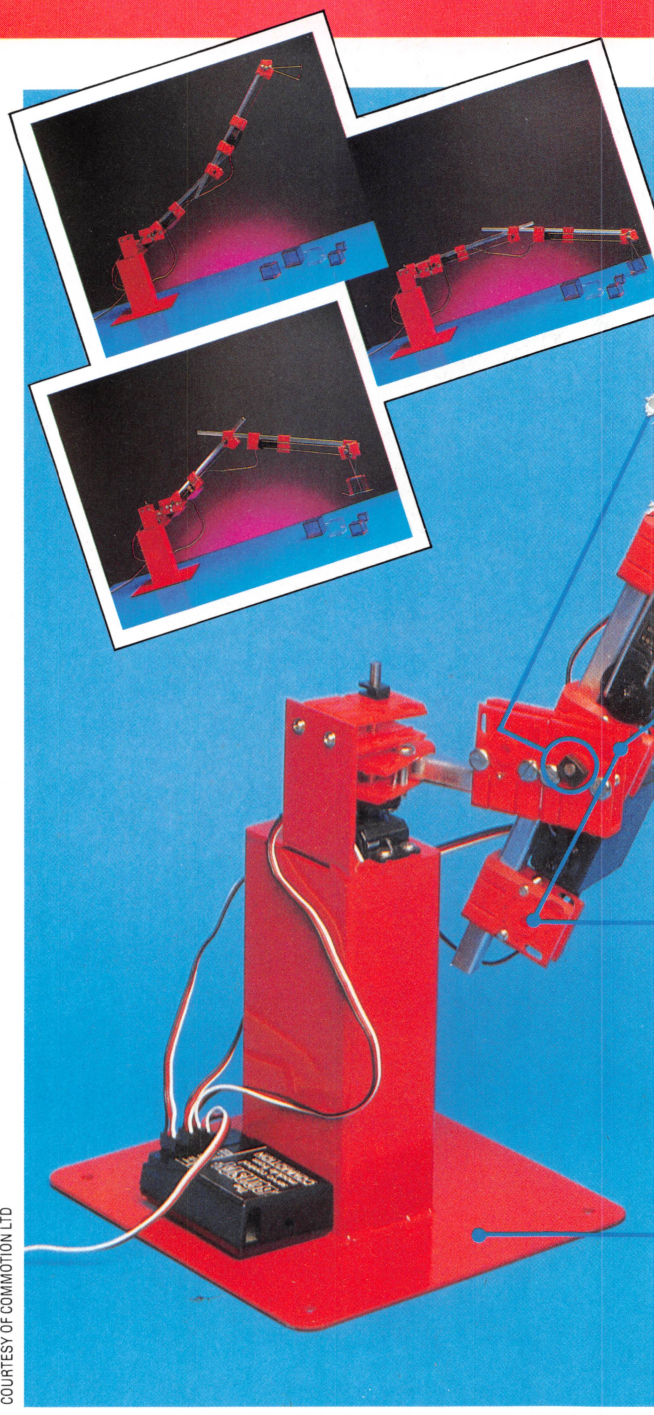
Successful computer systems design requires the analysis of a multitude of factors. Many of these factors are systems-based — including speed of processing, input/output management, and so on. But computers are used by humans, so systems designers must take human factors into account as well. The recent science of *human factors engineering* aims to incorporate traditional systems design and engineering with marketing and operational psychology to create a total man-machine system. Many new users of computers fear the power of the machines and feel intimated by them. To overcome this very deep-rooted emotional block, engineers design systems to be 'user friendly' and 'intuitive'. The use of menus and straightforward language in software, and the development of 'user interfaces' in hardware — such as Apple's mouse and icon system, or Hewlett-Packard's touch screen — aim to make the computer less frightening. A very recent development in human factors engineering finds interior designers working with systems design teams to create a total man-machine environment. The location of work surfaces, the positioning of computers, and other design elements all come together into a uniform system.

HYBRID INTEGRATED CIRCUIT

Large circuits can be assembled by combining a number of smaller circuits, each of which may be constructed by using a variety of technologies. The smaller components are placed on an insulating base material, then linked with conductive tracks that are printed on the base in several layers. The resulting circuit, which can then be connected to additional chips, transistors and other components, is called a *hybrid integrated circuit,* or 'hybrid IC' for short.

THE BEASTY WITHIN

Robotics is a topic in the forefront of developments in computing, but until recently robots have been used mainly in heavy industry. Now robot designers are turning their attention to the home computer owner. Budget-priced robots are gradually becoming available and the Beasty is one of the first.



COURTESY OF COMMOTION LTD

The Beasty is provided in kit form, and can be assembled by the user with the aid of a pair of screwdrivers. Also provided with the kit is the cassette-based software that contains the Robol operating system used to control the robot arm.

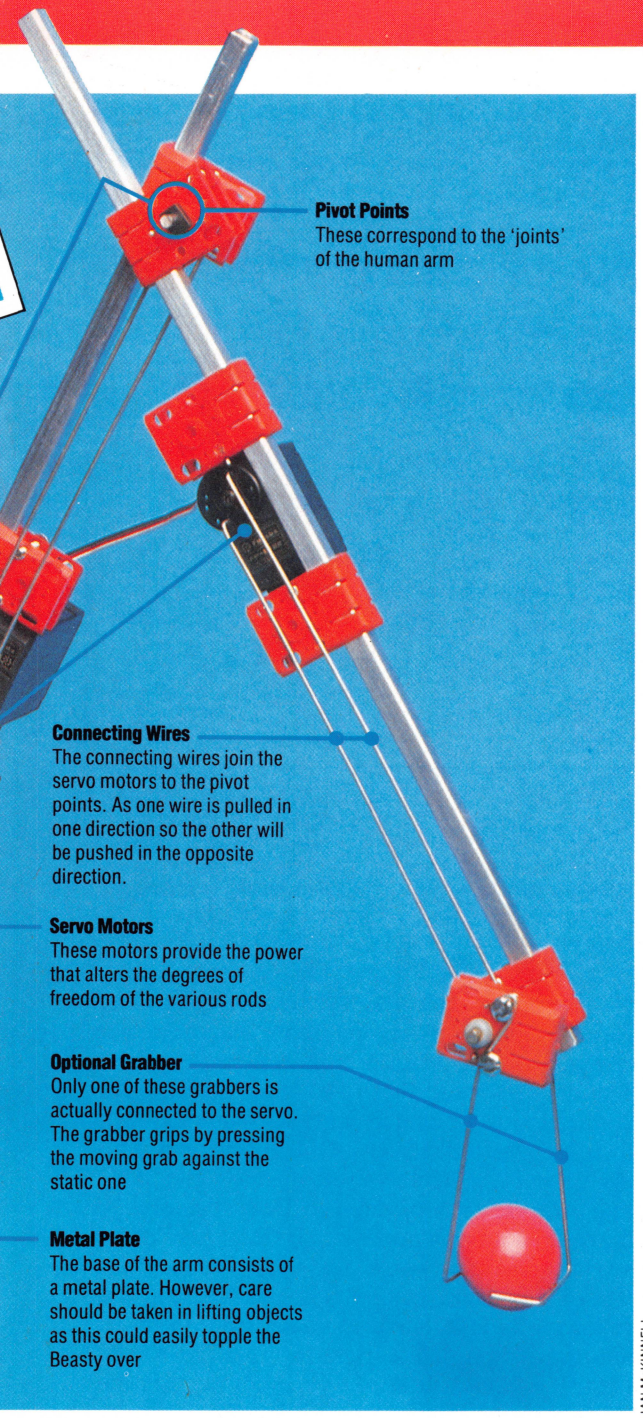
Two manuals are supplied. The construction booklet begins with a long introduction on the history of robotics, before going on to the actual construction details. The novice may find the sheer number of parts and the somewhat complicated instructions a little daunting; illustrations are provided, but these too could be more helpful. However, even the beginner will be able to put the arm together successfully — but it may take a little time to do so. As yet, it is not possible to buy the Beasty ready-made, although manufacturer Commotion claims that it will put one together if asked.

Once assembled, the Beasty consists of a base supporting a joint that allows lateral movement. To this joint is connected a short aluminium rod, which is attached to the upper part of the arm by a second joint. A third joint connects the forearm. These joints are powered by servo motors, each of which controls two short lengths of stiff wire, which are connected to the arm's 'skeleton'. As a servo motor turns it will pull one wire towards it and push the other in the opposite direction, thus turning the joint and moving the arm. A servo motor works by translating digital pulses into movement. The motor receives a series of pulses at a particular frequency, and these pulses are interpreted by the processor as an angle of movement. While the frequency remains constant, the motor will hold the arm in its current position; a change in the pulse frequency instructs the processor that a new angle is required and so the arm will move.

The FP128 servos used in the Beasty can generate 3.5kg/cm of pull. This means that at one centimetre along a shaft the servo is capable of lifting 3.5 kilograms, while at 10 centimetres along it can lift 350 grams. This is an important point to consider when weights are lifted — obviously, the servo at the 'shoulder', being furthest away from the weight to be lifted, will have the greatest strain placed on it.

The servo processor is housed in a small black box that is not attached to the arm itself. This box has sockets for the connection of up to four servo motors (the fourth connection is for an optional motor that may be used to operate a 'claw' or similar gripping device at the end of the forearm). There is also an input socket that interfaces with the BBC's user port, and a power lead that plugs into the auxiliary power socket of the computer.

Once the cassette software has been loaded, the screen displays a prompt to remind the user that the system is in Edit mode. A program line in Robol consists of a line number, a command, and a series of numbers, each of which corresponds to one of the four servo motor options. If the line contains the command MOVE, the numbers correspond to the frequency of the pulses that



IAN McKINNELL

maintain the servos in their current position. These numbers may be altered by the user, and while the system is in Edit mode the servo motor being adjusted will move as changes are made, allowing the arm to be positioned to the user's satisfaction. Once the operator is happy with the positions of each of the motors, Return must be pressed, whereupon a new Robol line is displayed and the next series of movements can be programmed.

The arm can be made to carry out a complete sequence of movements if the F0 function key is pressed. The program can be made to run from any line by first pressing F1 followed by F0 (this runs from the start of the program) or by changing the current line number by using the cursor keys. At the end of the program, the sequence of actions will automatically repeat. Should the user wish to stop the program, a MOVE instruction must be changed to STOP.

TIMED DELAYS

While executing a series of MOVE instructions, the arm can be made to pause by incorporating a WAIT instruction, followed by a number. This works by accessing the TIMER 1 pin of the user port, generating an interrupt. As the timer works in units of 1/100th second, WAIT 100 will produce a one-second delay before the next instruction is carried out.

The action of the arm can be greatly accelerated by changing the MOVE command to JUMP. Two timing statements are also included — JDELAY and MDELAY. The Beasty has a built-in delay that occurs before each line executes. This has a default value of 20 — i.e. 1/5th second — but this value may be changed by using JDELAY for JUMP statements and MDELAY for MOVE instructions.

The Robol operating system is easy to use, and it is a simple task to program the arm to perform complex movements within a matter of minutes. The operating manual is brief but perfectly adequate, although advanced programmers may find that it gives insufficient information for more complex programming. The Beasty may be controlled from BASIC by using the Driver program; this accesses the BBC Micro's user port in much the same way as the examples we have given in our Workshop series.

Commotion has also included a short program that allows backup copies of Robol software to be made. Unfortunately, most BBC disk drives use the auxiliary power socket on the BBC Micro, so the Beasty and a disk drive cannot be connected at the same time.

However, despite such minor niggles, the Beasty is certainly a worthwhile introduction to the field of robotics. It might be said that this is a device in search of an application, as the robot arm cannot really be said to be truly useful and will probably be purchased by only the most enthusiastic hobbyists. However, it is sure to be of value in schools and colleges, where it can be used to teach students the principles of robotic control.

THE BEASTY

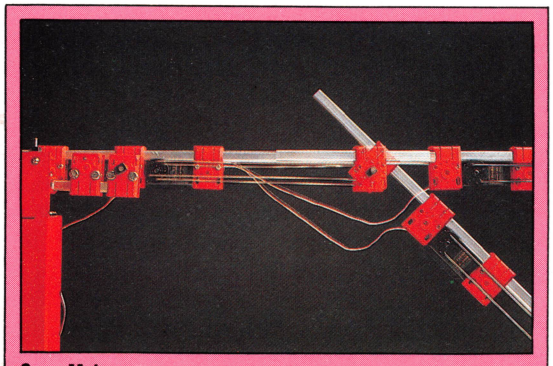
PRICE
£110
Interfaces connectors that plug into the user port and auxiliary supply of the BBC Micro.

SOFTWARE
Robol and driver routines available on cassette.

DOCUMENTATION
The Beasty comes with an assembly manual and a programming guide.

STRENGTHS
An easy to use and reasonably priced introduction to robotics.

WEAKNESSES
The assembly guide lacks clarity and there is a lack of obvious applications for the arm, although Commotion promised some upgrades that will widen its uses.



Servo Motors
The standard Beasty comes equipped with three servo motors connected to an interface box. Servo motors are popular motors for robotics because of the way they provide a constant force or 'torque' corresponding to the frequency of the digital signal with which they are supplied

LOST IN MAZES

In previous instalments of Workshop we developed the hardware and software to drive a two-motored vehicle and control its direction (see pages 585 and 612). Now we develop an 'intelligent' program that will steer our two-motor vehicle through a maze by selecting the shortest route.

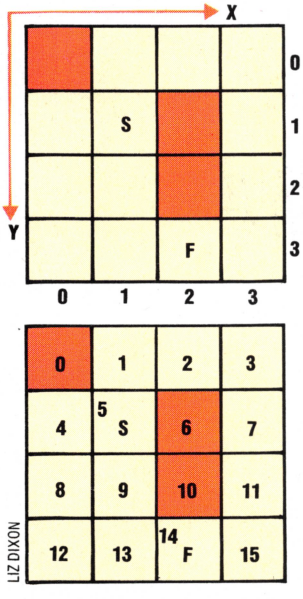
The first stage in constructing a maze is to decide on the area that will constitute the maze. This could be a table top or an area on the floor. The area designated should then be divided into a number of squares, the size of each square being dependent on the size of the vehicle that will be used to negotiate the maze. Each square should be large enough to allow the vehicle to pivot through 360° within a single square. The area can then be marked out as a grid. Objects such as books, cups, or short lengths of wood can then be placed in the area to form the maze.

The program requires you to specify the dimensions of the maze, and the locations of squares in the maze that are occupied and those that are free. The easiest method of doing this is to use a binary code: 1 indicating that a square is partially or fully occupied by an object, 0 indicating that the square is free. In order that the maze data does not have to be entered each time the program is run, this information must be written as a series of DATA statements. The final four items of data are the start and finish point co-ordinates. We can imagine the origin of the co-ordinate system as having its origin at the top left corner, the top row being row 0 and the leftmost column being column 0. This maze corresponds to the following DATA statements:

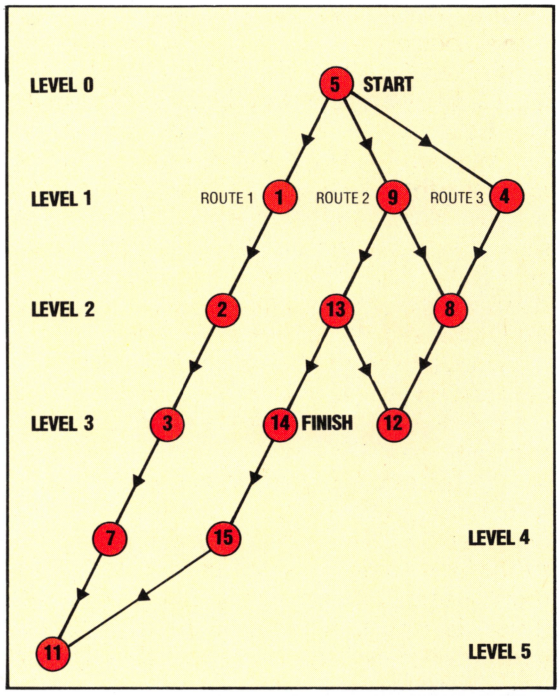
```
DATA 4,4:REM DIMENSIONS OF MAZE
DATA 1,0,0,0,0,0,1,0
DATA 0,0,1,0,0,0,0,0
DATA 1,1:REM COORDS OF START
DATA 2,3:REM COORDS OF FINISH
```

Finding a route through a maze does not present many difficulties. We can design a program that will trace a route from the start point, backing out of blind alleys and retracing steps until the finish point is eventually stumbled upon. The eventual route found (without the detours into blind alleys) may or may not be the shortest possible route. If we want to find the *best* route between the start and finish points then we must adopt a method that tests all possible routes between the two points. It is worth noting that our program interprets 'best route' as the route that uses the least number of squares.

Two-Way Drive
The maze solving program interprets the maze in two ways. As the maze is read in from data statements it is stored in a two-dimensional array, the start and finish points also being held initially as co-ordinates. In order to solve the maze the program must treat each square in the maze as a 'node' in a tree. Rather than using the initial co-ordinate system, each square is therefore numbered in order, starting at the top left-hand corner of the maze



LIZ DIXON



Tree Structure
Before the best route through the maze can be found, a 'tree' representing the relationships between squares within the maze has to be constructed. Each node is considered in turn, creating lower levels of nodes. Level 1 nodes are one square removed from the start; level 2 nodes are two squares removed from the start, and so on. It is fairly straightforward for us to draw the tree, but implementing this structure in BASIC is more difficult

We can make the task of testing each route easier by enforcing a structure on the maze data that represents relationships between squares. The data structure that most lends itself to this application is a hierarchical tree. Beginning with the start point as the 'root' of the tree, we can build up a second generation of squares (or 'nodes') that are one square removed from the root. A third generation of nodes can be built from these second generation nodes, and so on. We can draw a tree for any maze by numbering each square and following the rule that descendants from any node are drawn from left to right in the order North, East, South and West of the parent node in the original maze.

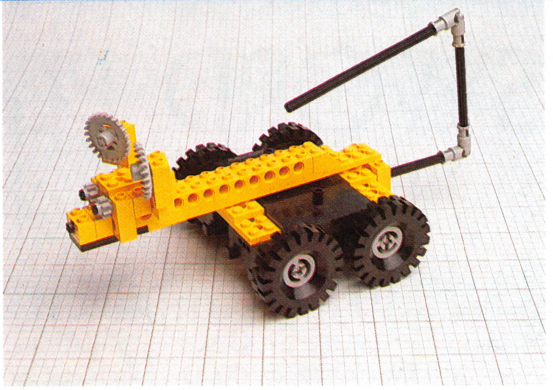
This simple maze can be solved in five ways, without retracing one's steps. Three possible solutions are shown above, as routes through the tree and as actual routes through the maze. It is obvious to us that route 2 is the shortest route, but this is because we are able to evaluate the tree laterally; that is we can consider the maze as a whole. The computer must solve the tree in a linear way, taking each possible route systematically until the finish node is found or a blind alley is reached. In the first case a record of the sucessful route must be kept; in the second the path taken must be marked as a blind alley before restarting from the root node. The program will continue to travel through the tree until all branches from the root node have been tried.

BASIC does not lend itself readily to search algorithms of this type and the programming can

often appear clumsy and unwieldy. Languages such as LOGO and ALGOL are much better equipped to carry out this sort of task. In BASIC we have two main tasks to carry out. Firstly we must derive our tree from the maze data, as presented to the program. And for each square of the maze we must have four pointers showing which square lies in each of the four directions. The best way to store this pointer system is in a two-dimensional array, TR(N,D), where N is the square number and D is the direction 1 to 4. Thus in our simple maze, TR(9,1) would be 5— the square lying to the north of square 9. When the square in a particular direction is not free, or there is a boundary to the maze then this can be marked by a special value, for example -1.

As the tree is negotiated the route taken is stored in a pseudo-stack, implemented using a one-dimensional array and a variable, D, to point to the next available space on the stack. The shortest route encountered at any time is also stored in a one-dimensional array, with the number of steps for the route stored in the first element of the array.

When the program has worked its way through the tree, a record of the best route will be held as a series of square numbers. On the assumption that the vehicle originally faces north in the start square, it can be directed using the simple mathematical relationships between the



IAN McKINNELL

direction to be travelled and the difference between two consecutive square numbers in the route array. For example, in our simple maze, a difference of +4 would indicate north, -4 indicate south, and so on. We must then calculate the angle to be turned through to change direction, before proceeding one square forwards. As the vehicle uses simple DC electric motors, turning angles and distances travelled are governed by the length of time that a particular combination of motors is on for. To make practical use of the program some initial experiments need to be done to determine the time intervals required to turn through 90° and to advance one square. This information should be entered in the variables AF and FF, respectively. The BBC version requires units of 1/100th of a second, the Commodore 64 version requires 1/60th of a second units.

Solving The Maze

```
900 REM ******************
910 REM ******************
920 REM **                **
930 REM ** CBM 64 MAZE   **
950 REM **                **
960 REM ******************
970 REM ******************
980 :
990 REM **** MAIN CALLING PROGRAM ****
1000 GOSUB 1600:REM READ MAZE DATA
1100 GOSUB3700:REM PRINT MAZE
1200 GOSUB5400:REM CONSTRUCT TREE
1210 GOSUB7700:REM TRAVERSE TREE
1215 IF S(0)=9999 THEN PRINT "NO SOLUTION":END
1220 GOSUB8200:REM DIRECT VEHICLE
1400 END
1500 :
1600 REM **** READ MAZE DATA/INITIALISE ****
1700 READ SX,SY
1800 DIM MZ(SX,SY),TR(SX*SY,4),DR(4),RT(SX*SY),
     LN(SX*SY),CN(SX*SY)
1900 FOR Y=0 TO SX-1
2000 FOR X=0 TO SX-1
2100 READ A:MZ(X,Y)=A
2200 NEXT X,Y
2300 :
2400 READ XS,YS,XF,YF
2500 DATA 4,4
2600 DATA 1,0,0,0,0,0,1,0
2700 DATA 0,0,1,0,0,0,0,0
2800 DATA 1,1:REM START COORDS
2900 DATA 2,3:REM FINISH COORDS
3000 :
3100 DR(1)=-SX:DR(2)=1:DR(3)=SX:DR(4)=-1
3110 ID(1)=3:ID(2)=4:ID(3)=1:ID(4)=2
3120 SR(0)=9999:REM INIT SHORTEST ROUTE
3130 DDR=56579:DATREG=56577:POKE DDR,255
3140 AF=30:FF=45:REM ANGLE AND FWRD TIME FACTORS
3200 FOR I=1 TO 25:CD$=CD$+CHR$(17):NEXT I
3500 RETURN
3600 :
3700 REM **** PRINT MAZE ****
3800 PRINTCHR$(147):REM CLEAR SCREEN
3900 FOR X=0 TO SX-1
4000 FOR Y=0 TO SY-1
4100 GOSUB 4900:REM POSITION CURSOR
4200 PRINT CHR$(32+MZ(X,Y)*134)
4300 NEXT Y,X
4400 :
4500 X=XS:Y=YS:GOSUB4900:PRINT"S"
4600 X=XF:Y=YF:GOSUB4900:PRINT"F"
4700 RETURN
4800 :
4900 REM **** POSITION CURSOR AT X,Y ****
5000 PRINT CHR$(19);:PRINTTAB(X)LEFT$(CD$,Y);:RETURN
5300 :
5400 REM **** CONSTRUCT TREE ****
5500 :
5600 REM ** INITIALISE WITH TERMINATORS ****
5700 FOR P=0 TO SX*SY-1:FORI=1TO4:TR(P,I)=-1:NEXT I,P
6100 :
6110 REM ** CALCULATE START & FINISH ****
6120 X=XS:Y=YS:GOSUB9200:S=N
6130 X=XF:Y=YF:GOSUB9200:F=N
6140 :
6200 REM ** CONSTRUCT **
6300 LC=1:CC=0:REM INIT STACK PTRS
6400 LN(LC)=S:REM START POINT
6450 CN=LN(LC):REM GET CURR NODE OFF STACK
6500 DF=0
6600 FOR D=1 TO 4
6700 N=CN+DR(D):GOSUB 8900:REM CONVERT TO X,Y
6800 IF(X<0 OR X>SX-1 OR Y<0 OR Y>SY-1)THEN 7300
6900 IF MZ(X,Y)=1 THEN 7300
7000 IF(D=2 AND N/SX=INT(N/SX))THEN7300
7100 IF(D=4 AND CN/SX=INT(CN/SX))THEN7300
7200 IF TR(N,ID(D))=CN THEN 7300
7210 TR(CN,D)=N:DF=1
7220 CC=CC+1:CN(CC)=N:REM PUSH ONTO CURRENT STACK
7300 NEXT D
7310 IF(DF=0 AND LC=1) THEN RETURN:REM TERMINAL NODE
7320 :
7330 LC=LC-1:REM DEC LAST STACK PTR
7340 IF LC>0 THEN 6450:REM NEXT NODE
7345 :
7350 REM ** COPY CURR. STACK TO LAST STACK **
7360 FOR I=1 TO CC:LN(I)=CN(I):NEXT I
7390 LC=CC:CC=0:GOTO6450:REM NEXT NODE
7600 :
7700 REM **** TRAVERSE TREE ****
7720 C=0:RN=S:CN=RN:EF=0
7730 C=C+1:RT(C)=CN
7740 IF CN=F THEN GOSUB8100:GOSUB8000:IF EF=0 THEN 7730
7745 IF EF=1 THEN RETURN
7750 DF=0
7760 FOR D=1 TO 4
7770 IF TR(CN,D)<>-1 THEN CN=TR(CN,D):DF=1:DR=D:D=4
7780 NEXT D
7790 IF DF=0 THEN GOSUB8000
7800 IF EF=0THEN 7730
7810 IF EF=1 THEN RETURN
7820 :
8000 REM **** RESTART AT ROOT ****
8010 TR(RT(C-1),DR)=-1
8020 CN=RN:C=0
8030 IF(TR(CN,1)ANDTR(CN,2)ANDTR(CN,3)ANDTR(CN,4))=-1 THEN EF=1
8040 RETURN
8050 :
8100 REM **** SAVE ARRAY ****
8110 IF C>SR(0) THEN RETURN:REM IS NEW ROUTE SHORTER?
8120 SR(0)=C
8130 FOR I=1 TO C:SR(I)=RT(I):NEXT I:RETURN
8140 :
8200 REM **** DIRECT VEHICLE ****
8205 PD=1:REM ASSUME INITIAL
     DIRECTION NORTH
8210 FOR C=1 TO SR(0)-1
8220 DF=SR(C+1)-SR(C)
8222 :
8225 REM ** FIND REQUIRED DIRECTION **
8230 FOR I=1 TO 4
8240 IF DF=DR(I) THEN D=I:I=4
8250 NEXT I
8260 :
8265 DR=D-PD:PD=D
8270 H=INT(4+DR/4):R=(4+DR)-4*H
8275 :
8277 REM ** DO TURN **
8280 FOR I=1 TO R
8290 POKE DATREG=9:REM CLOCKWISE TURN
8300 T=TI
8310 IF (TI-T)<AF THEN 8310:REM WAIT
8320 POKE DATREG=0:REM OFF
8330 NEXT I
8340 :
8350 REM ** FORWARD **
8360 POKE DATREG=5
8370 T=TI
8380 IF (TI-T)<FF THEN 8380
8390 POKE DATREG=0
8400 NEXT C
8410 :
8420 RETURN
8430 :
8900 REM **** CONVERT N TO X,Y ****
9000 Y=INT(N/SX):X=N-SX*Y:RETURN
9200 :
9210 REM **** CONVERT X,Y TO N ****
9220 N=Y*SX+X:RETURN
```

For The BBC

Make the following changes:

```
3130 DDR=&FE62:DATREG=&FE60

8290 ?DATREG=9
8300 TIME=0
8310 REPEAT UNTIL TIME>=AF
8320 ?DATREG=0

8360 ?DATREG=5
8370 TIME=0
8380 REPEAT UNTIL TIME>=FF
8390 ?DATREG=0
```

SWORD PLAY

The list processing facilities of LOGO make it ideal for a variety of games applications. Here, we show you how the language can be used in the development of a text-based adventure game. We approach the implementation in a general way, to allow you to build up your own game by defining the locations, objects and perils yourself.

In this article, we'll restrict ourselves to looking at the more general aspects of programming an adventure game and deal with the specific details for a particular game in the next instalment.

In all adventure games there are five basic activities that the player should be able to perform: you need to pick up objects or drop them, to list the things you are carrying, to look at your surroundings and to move about the game from room to room (or location to location). So it is these basic commands that we will program first of all. For simplicity, we will restrict the form of the commands to one of two types: either single words (such as LOOK) or verb-noun pairs (such as DROP RING). The program will maintain two lists: one called INVENTORY, which will be a list of everything the player is currently carrying, and the other, called simply CONTENTS, will be a list of the objects in the current room.

The first command we will define is INVENTORY:

```
TO INV
  PRINT [YOU ARE CARRYING:]
  IF EMPTY? :INVENTORY THEN PRINT [NOTHING]
  ELSE PRINT :INVENTORY
END
```

Notice that this procedure uses the full form of the IF statement: IF ‹condition› THEN ‹action1› ELSE ‹action2›. The command for picking up an object will be GET:

```
TO GET :ITEM
  IF MEMBER? :ITEM :CONTENTS
  THEN GETIT :ITEM ELSE PRINT [I
  CAN'T IT'S NOT HERE]
END
```

MEMBER? is a primitive that tests to see if an element belongs to a list. To 'get' an item we need to do two things: add it to the inventory and remove it from the list of contents. These are the procedures that do these tasks:

```
TO GETIT :ITEM
  ADD.TO.INV :ITEM
  REMOVE.FROM.ROOM :ITEM
END

TO ADD.TO.INV :ITEM
  MAKE "INVENTORY SENTENCE :ITEM
  :INVENTORY
END

TO REMOVE.FROM.ROOM :ITEM
  MAKE "CONTENTS DELETE :ITEM
  :CONTENTS
END
```

The last of these procedures involves deleting an element from a list — which was one of the exercises given in the previous instalment.

```
TO DELETE :ITEM :LIST
  IF :ITEM = FIRST :LIST THEN OUTPUT BUTFIRST
  :LIST
  OUTPUT SENTENCE FIRST :LIST DELETE :ITEM
  BUTFIRST :LIST
END
```

The command for dropping an object is implemented in a similar way:

```
TO DROP :ITEM
  IF MEMBER? :ITEM :INVENTORY THEN DROPIT
  :ITEM ELSE PRINT [YOU DON'T HAVE IT TO
  DROP!]
END

TO DROPIT :ITEM
  REMOVE.FROM.INV :ITEM
  ADD.TO.ROOM :ITEM
END

TO REMOVE.FROM.INV :ITEM
  MAKE "INVENTORY DELETE :ITEM :INVENTORY
END

TO ADD.TO.ROOM :ITEM
  MAKE "CONTENTS FPUT :ITEM :CONTENTS
END
```

Having entered all the procedures we have given so far, it is now time to test their operation. First of all, we must define the two global variables — INVENTORY and CONTENTS — and then test for the following commands:

```
MAKE "CONTENTS [SWORD SPEAR TORCH]
MAKE "INVENTORY [LANTERN]
GET "SWORD
DROP "LANTERN
```

Now examine CONTENTS and INVENTORY using these statements:

```
PRINT :CONTENTS
PRINT :INVENTORY
```

and check that they are correct.

Notice that we used quotation marks before the

names of the objects in both the GET and DROP commands. Using quotes this way might be second nature to a LOGO programmer, but it is likely to be very confusing for an adventurer who knows nothing about the language. In order to allow the more natural GET SWORD to be used, we must define SWORD as follows:

```
TO SWORD
   OP "SWORD
END
```

Of course, we'll have to do this for each noun used in the game.

The command LOOK will print a description of the current room, a list of its contents, and the possible exit routes from the room. To do this we'll need two further lists — a description list and an exit list. In order to allow for fairly long descriptions taking up more than one line across the screen, the description list is defined as a list of lists. For example:

```
MAKE "DESCRIPTION [[YOU ARE STANDING AT THE
ENTRANCE][TO A CAVE]]
```

To keep a record of how the rooms join each other, every room is assigned a number. The exit list is simply a list of sublists, each consisting of a direction and a room number. Thus:

```
MAKE "EXIT.LIST [[N 4][E 6]]
```

We can now define LOOK:

```
TO LOOK
   PRINTL :DESCRIPTION
   PRINT "
   PRINT [YOU CAN SEE:]
   IF EMPTY? :CONTENTS THEN PRINT [NOTHING
   SPECIAL] ELSE PRINT :CONTENTS
   PRINT "
   PRINT [YOU CAN GO:] PRINT.EXITS :EXIT.LIST
   PRINT "
END
```

Two special print routines have been used in this procedure to make the display easier to read. PRINTL is used to print several lines of text.

```
TO PRINTL :LIST
   IF EMPTY? :LIST THEN STOP
   PRINT FIRST :LIST
   PRINTL BUTFIRST :LIST
END
```

PRINT.EXITS prints the exits from the room without printing the room numbers.

```
TO PRINT.EXITS :LIST
   IF EMPTY? :LIST THEN PRINT " STOP
   MAKE "EXIT FIRST :LIST
   PRINT1 FIRST :EXIT
   PRINT1 "' '
   PRINT.EXITS BUTFIRST :LIST
END
```

We can describe everything that is known about a room in the game by putting together the three sublists: the description, the contents and the exits. For example:

```
MAKE "ROOM.1 [[[YOU ARE STANDING AT THE
ENTRANCE][TO A CAVE]] [SWORD][[N 4][E 6]]]
```

Given that ROOM.1 is defined in this way, we could split it into its individual components with the following procedure:

```
TO ASSIGN.VARIABLES
   MAKE "ROOM THING "ROOM.1
   MAKE "DESCRIPTION DESCRIPTION :ROOM
   MAKE "CONTENTS CONTENTS :ROOM
   MAKE "EXIT.LIST EXIT. LIST :ROOM
END
```

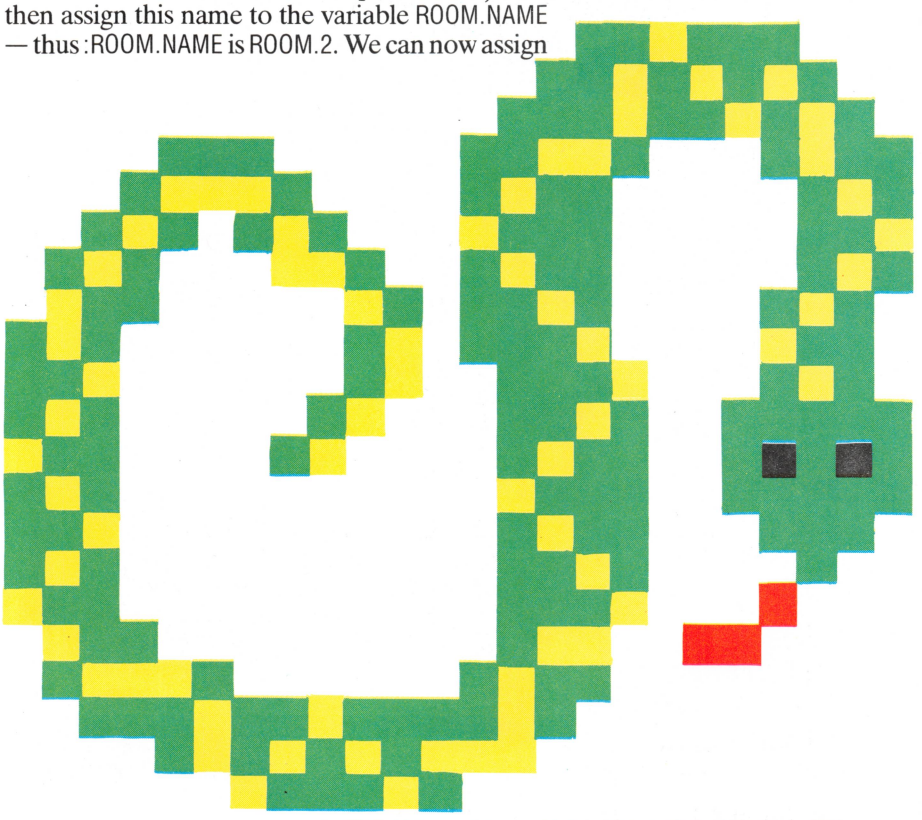
THING "ROOM.1 is an alternative to :ROOM.1; it means 'the contents of the variable ROOM.1'. We will discuss the reason for using this form shortly. The subprocedures are defined as follows:

```
TO DESCRIPTION :ROOM
   OUTPUT ITEM 1 :ROOM
END

TO CONTENTS :ROOM
   OUTPUT ITEM 2 :ROOM
END

TO EXIT.LIST :ROOM
   OUTPUT ITEM 3 :ROOM
END
```

As it stands, this procedure works for ROOM.1 only. We need to extend it so that it can be used more generally for any room. We do this by using a global variable, HERE, which contains the number of the current room. Let's say it is 2 at the moment. The LOGO primitive WORD outputs a word consisting of a combination of its two inputs (thus, WORD "ROOM. :HERE would output ROOM.1). We then assign this name to the variable ROOM.NAME — thus :ROOM.NAME is ROOM.2. We can now assign



```
ROOM as THING :ROOM.NAME.
  TO ASSIGN.VARIABLES
    MAKE "ROOM.NAME WORD "ROOM. :HERE
    MAKE "ROOM THING :ROOM.NAME
    MAKE "DESCRIPTION DESCRIPTION :ROOM
    MAKE "CONTENTS CONTENTS :ROOM
    MAKE "EXIT.LIST EXIT.LIST :ROOM
  END
```

It is now possible for you to draw up a map of the locations in your adventure game, and list the descriptions of them (with contents and exits). In the next instalment, we will conclude our general discussion by looking at movement between locations and how 'perils' are implemented. We will then begin considering a complete adventure game as an example of what can be done.

Logo Flavours

Some versions of MIT LOGO do not have EMPTY?, ITEM, COUNT or MEMBER?. Definitions for the first three were given in the last instalment. The definition of MEMBER? is:

```
TO MEMBER? :ITEM :LIST
  IF :LIST = [] THEN OUTPUT "FALSE
  IF :ITEM = FIRST :LIST THEN OUTPUT "TRUE
  OUTPUT MEMBER? :ITEM BUTFIRST :LIST
END
```

In all LCSI versions use:

```
EMPTYP for EMPTY?
LISTP for LIST?
MEMBERP for MEMBER?
TYPE for PRINT1
```

There is also a primitive, EQUALP, which tests whether its two inputs are the same. Use this for comparing lists and words in place of =. (The equals sign works for lists on some LCSI versions, but not on others.)

The full IF syntax in LCSI LOGO is demonstrated by this example:

```
IF EMPTYP :CONTENTS [PRINT [NOTHING
  SPECIAL]] [PRINT :CONTENTS]
```

The first list after the condition is performed if the condition is true, and the second if it is false.

Answers

1. For printing in reverse order:

```
TO PRINTR :LIST
  IF EMPTY? :LIST THEN PRINT " STOP
  PRINT1 LAST :LIST
  PRINT1 "' '
  PRINTR BUTLAST :LIST
END
```

The following procedure outputs the list in reverse order, rather than printing it.

```
TO REVERSE :LIST
  IF EMPTY? :LIST THEN OUTPUT []
  OUTPUT SENTENCE LAST :LIST REVERSE
  BUTLAST :LIST
END
```

2. We give a procedure to delete an item from a list in the main text of this instalment.

Logo Poetry

In the previous instalment, we set you the problem of improving LOGO's poetry writing abilities. One possible method is to create a 'template' of a sentence structure (such as 'noun, verb, noun') and choose words from sublists of these parts of sentences. For example, using this template:

```
POEM2 [NOUN VERB NOUN]
```

we could have:

```
TO POEM2 :TEMPLATE
  IF EMPTY? :TEMPLATE PRINT "STOP
  POEM2.1 FIRST :TEMPLATE
  POEM2 BUTFIRST :TEMPLATE
END

TO POEM2.1 :WRD
  IF :WRD="NOUN (PRINT1"' ' GETRANDOM
  :NOUN)
  IF :WRD="VERB (PRINT2"' ' GETRANDOM
  :VERB)
END
```

This method will become very cumbersome if we introduce many more parts of speech. Let's take a closer look at variables again to see if we can improve on this. First of all, enter:

```
MAKE "ROSE "SWEET
MAKE "OTHERNAME "ROSE
```

Now you will find that:

```
PRINT :ROSE prints SWEET
PRINT :OTHERNAME prints ROSE
PRINT THING :OTHERNAME prints SWEET
```

THING gives us the value associated with a name. In the last case, the name following THING is the value of the variable OTHERNAME — in other words, ROSE. Using this idea, we can rewrite our poem procedure:

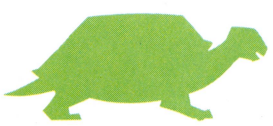
```
TO POEM2.1
  (PRINT1 "' ' GETRANDOM THING :WRD)
END
```

The procedure call POEM2.1 "NOUN assigns the value NOUN to the variable WRD. THING :WRD is then the value associated with NOUN — the list of nouns. Using the template:

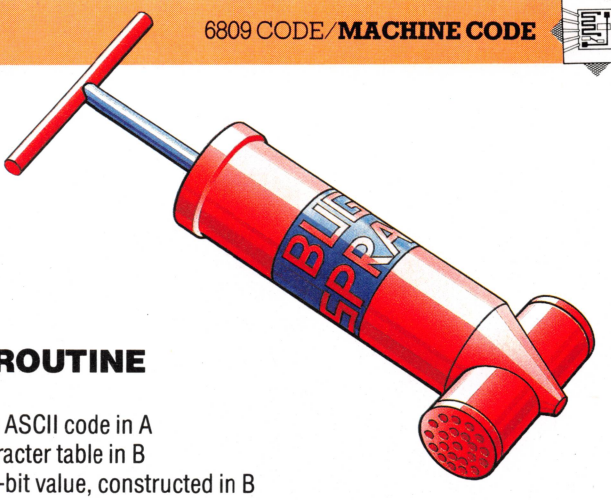
```
POEM2 [ART ADJ NOUN ADV VERB PREP ART
  ADJ NOUN]
```

together with a suitable choice of vocabulary, we obtained the following output:

```
A GREEN PLANET LOUDLY SPUN UNDER A
  PARANOID DOME
A DILAPIDATED SHIP SLOWLY FLEW TOWARDS
  THE SPLENDID PLANET
```



PEST CONTROL



We continue to develop the debugger program, which we began on page 758. First, we must complete the set of routines that we need for the module that handles input and output, and then construct the module itself.

There are four more routines to develop for the I/O module: GETHX2, GETHX4, PUTHEX and PUTCR. The first two processes are used to get hexadecimal digits from the keyboard: GETHX2 gets a two-digit hex number and GETHX4 gets a four-digit number. The first thing we must do, when designing these routines, is decide whether we will insist on two or four digits always being entered (which is easier to program but less user-friendly), or allow fewer characters followed by a Return. A further problem is whether to allow the use of a backspace character to delete characters already entered.

We will use the simplest method for the GETHX4 routine: four digits must be entered, and the backspace will not be allowed. The 16-bit value (signifying an address) can be returned in the D register.

GETHX2 is more of a problem if we consider the circumstances in which it will be used. Eight-bit quantities will have to be entered for the function to inspect and change memory (command M), which involves accessing an address. The contents of this address are displayed, and the user may then enter a Return (to move on to the next location in sequence) or a two-digit hex number (which will be stored at that location) or some other character (a dot, for example, to exit to the command level). We can add the two extra valid characters to the end of the string of valid hex digits. GETHX2 must then accept either two hex digits or a Return or a dot. The eight-bit value can be returned in B and we must use A to indicate which of these situations has arisen. A will have the value 0 if a two-digit number was entered, 1 if a Return was entered, or –1 if a dot was entered. These values enable us to test the value in A without having to compare it with another value.

Let's assume for the moment that the following declarations have been made for this module:

```
HEXCHS    FCC'0123456789ABCDEF'
DOT       FCB '.
RETURN    FCB 13 (ASCII code for Return)
```

We can pass 16 as the length of the string for GETHX4, where we only need the hex digits, and 18 as the length for GETHX2, where we need the other two characters as well.

THE GETHX2 ROUTINE

Data:

Next-Character is the ASCII code in A
Offset into Valid-Character table in B
Hex-Value is an eight-bit value, constructed in B
Flag is either 0, 1 or –1 in A

Process:

```
Get Next-Character
IF Character is a dot (Offset = 16) then
   Set Flag to –1
ELSE if character is a Return (Offset = 17) then
   Set Flag to 1
ELSE
   Save Offset temporarily
   Get Next-Character (hex digits only valid at
   this point)
   Construct Hex-Value
ENDIF
```

The final coded form of GETHX2 is given on page 779. The coding of the GETHX4 routine is now made slightly easier by using parts of this routine. By making HX4 an alternative entry point to the GETHX2 routine, we can call that routine and ensure that only valid hex digits are accepted — provided we load B with 16 before the call. Thus, the process for getting four hex digits is made considerably less complex.

THE GETHX4 ROUTINE

Data:

Hex-Number is a 16-bit value to be returned in D
Most-Significant-Byte and
Least-Significant-Byte are both eight-bit values to be returned in B

Process:

```
Get Most-Significant-Byte
Save Most-Significant-Byte temporarily
Get Least-Significant-Byte
Construct Hex-Number
```

The routine is given, in its final form, after the GETHX2 code.

The routines for displaying characters are less complicated to design. For the PUTHEX routine, we will assume that the eight-bit number we require is to be found in B.

THE PUTHEX ROUTINE

Data:

Number is the eight-bit value found in B
Offset is the four-bit offset put into HEXCHS

Process:

```
Extract most significant four bits of Number as
  Offset
```

Display HEXCHS (Offset)
Extract least significant four bits of Number
Display HEXCHS (Offset)

The final routine needed for handling input and output is the PUTCR subroutine. This is straightforward, and the final coded form is self-explanatory. Having coded all the necessary routines, we can now design the I/O module itself.

THE INPUT/OUTPUT MODULE

Process:

GetCommand will return Offset in B, which can be used as an offset into a jump table
GetAddress leaves return address in D
GetValue leaves return value in B, flag in A
DisplayValue is passed in B
DisplayAddress is passed in D

The final coded form of the I/O module is given on the following page. Now we can return to the Breakpoint module that we began in the last instalment (see page 758). We have already given the code for the second process in this module, which sets up breakpoints. We put aside the problem of coding the first process (inserting breakpoints) because it involved getting an address. Having now dealt with this task in the routines given here, we can proceed to give the coded version of the process — which incorporates a branch to the GETADD subroutine.

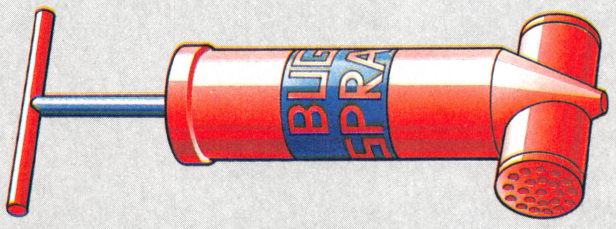
Notice that in the code, the command INC NUMBP, PCR adds 1 to the Number-Of-Breakpoints. At this point, A is one less than the Number-Of-Breakpoints, which is the correct offset into the breakpoint table. However, the address is returned in D, and this is going to destroy

The GETHX2 Routine

```
GETHX2  LDB    #18          Number-Of-Valid-Chars
HX4     PSHS   X            Save used register
        LEAX   HEXCHS,PCR   Get address of Valid-Chars in X
        BSR    GETCH        Get Next-Character
IFOO    CMPB   #16          If Offset = 16
        LDA    #$FF         Set flag to -1 (in two's
        BRA    ENDFOO       complement)
        CMPB   #17          If Offset = 17
        LDA    #1           Set flag to 1
        BRA    ENDFOO
        LSLB                Shift B left four places to form
        LSLB                most significant digit; B holds
        LSLB                offset in HEXCHS and hence the
        LSLB                binary value
        PSHS   B            Save B temporarily
        LDB    #16          Only hex digits now valid
        BSR    GETCH        Next-Character
        ADDB   1,S+         Construct eight-bit number and
        PULS   X,PC         lose temporary B
```

The PUTCR Routine

```
PUTCR   PSHS   A            Save A
        LDA    #13          ASCII code for Return
        BSR    OUTCH        Display it
        PULS   A,PC
```

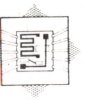


The GETHX4 Routine

```
GETHX4  LDB    #16
        BSR    HX4          Get Most-Significant-Byte
        PSHS   B            Save Most-Significant-Byte
        LDB    #16          temporarily
        BSR    HX4          Get Least-Significant-Byte in B
        PULS   A            Get Most-Significant-Byte back in A
        RTS                 Required value is in D
```

Display-Breakpoint Routine

```
BPLABS  FCC    '1 2 3 4 5 6 7 8 9101112131415 16'
SPACE   FCB    32           ASCII code for a Space
DISPBP  PSHS   A,B,X,Y
        LEAX   BPTAB,PCR    Address of Breakpoint-Table
        LEAY   BPLABS,PCR   Address of labels
        CLRB                Set Breakpoint-Number to zero offset
WHIL01  CMBP   NUMBP,PCR    While Breakpoint-Number <=
        BGT    ENDW01       Number-Of-Breakpoints
        LDA    ,Y+          Display label
        BSR    OUTCH
        LDA    ,Y+
        BSR    OUTCH
        LDA    SPACE,PCR    Display a Space
        BSR    OUTCH
        PSHS   B            Save B temporarily
        LDD    ,X++         Save Address
        BSR    DSPADD
        PULS   B            Restore B
        BRA    WHIL01
ENDW01  PULS   A,B,X,Y      Restore and return
```



the value in A because the D register comprises the A and B registers. Therefore, we use Y to hold the actual address.

Having coded the first Breakpoint process, there are three processes remaining. Two of these reverse the two processes we have coded so far: Uninsert-Breakpoint will remove a breakpoint from the table, and Unset-Breakpoint takes out the SWI-Opcode and puts back the original value. These two routines will be looked at in the next instalment. The third remaining routine, to display all the breakpoints, is the last routine that we will code here.

DISPLAY-BREAKPOINTS

Data:

Breakpoint-Number is an eight-bit counter to run through the Breakpoint table in B
Current-Breakpoint is the address to be displayed
Breakpoint-Labels are two-digit (decimal) numbers to label the addresses as they are displayed
Space is the space character that separates a label from an address

Process 3: Display-Breakpoints

```
Set Breakpoint-Number to 1 (an actual offset of
zero)
While Breakpoint-Number <= Number-Of-
Breakpoints
    Display Breakpoint-Labels(Breakpoint-
        Number)
    Display Breakpoint-Table(Breakpoint-Number)
    Increment Breakpoint-Number
Endwhile
```

End of Process 3

The PUTHEX Routine

```
PUTHEX  PSHS   A,B,X        Save used registers
        PSHS   B            Save B temporarily
        LEAX   HEXCHS,PCR   Address of HEXCHS in X
        LSRB                Four right shifts for most
        LSRB                significant four bits
        LSRB
        LSRB
        LDA    B,X          Get appropriate character in A
        BSR    OUTCH        Display it
        PULS   B            Get B back
        ANDB   #%00001111   Mask off most significant four
                            bits
        LDA    B,X          Second character
        BSR    OUTCH
        PULS   A,B,X,PC     Restore and return
```

Insert-Breakpoint Routine

```
BP01    PSHS   A,B,X,Y      Save used registers
        LDX    BPTAB        Address of Breakpoint-Table
        LDA    NUMBP,PCR
IF01    CMPA   MAXBP,PCR    If Number-Of-Breakpoints <
        BGE    ENDF01       Max
        INC    NUMBP,PCR    Add 1 to Number-Of-
                            Breakpoints
        LSLA                Multiply Offset by two for 16-bit
        LEAY   A,X          table
        BSR    GETADD       Get the address
        STD    ,Y           Store address in Breakpoint-
                            Table
ENDF01  PULS   A,B,X,Y      Restore and return
```

The Input/Output Module

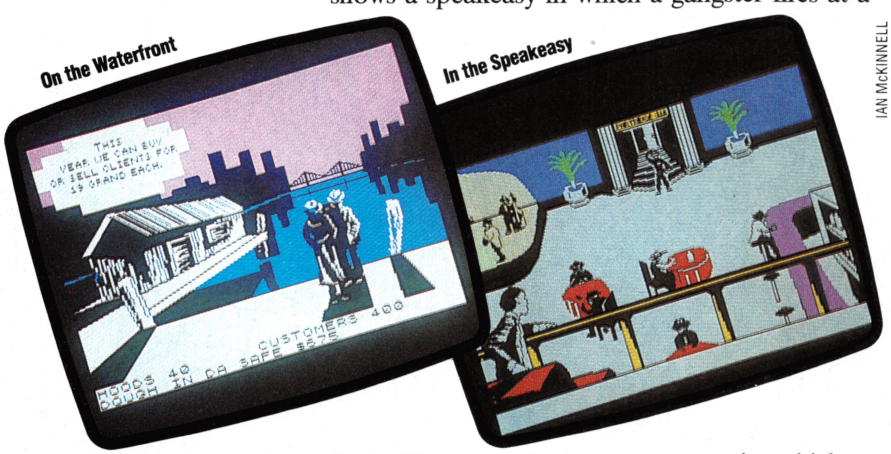
```
HEXCHS  FCC    '0123456789ABCDEF'
DOT     FCB    '.
RETURN  FCB    13           ASCII code for Return
COMNDS  FCC    'BUDSGRMQ'
GETCOM  PSHS   A,X          Save A and X contents
        LEAX   COMNDS,PCR   Address of command characters
                            in X
        LDB    #8           Number-Of-Valid-Characters
        BSR    GETCH        Get Character
        PULS   A,X,PC       Return
GETADD  BSR    GETHX4
        BSR    PUTCR
        RTS
GETVAL  BSR    GETHX2
        BSR    PUTCR
        RTS
DSPVAL  BSR    PUTHEX
        BSR    PUTCR
        RTS
DSPADD  PSHS   B            Save B temporarily
        TFR    A,B          Most-Significant-Byte in B
        BSR    PUTHEX
        PULS   B            Retrieve B
        BSR    PUTHEX
        PULS   B
        BSR    PUTHEX
        BSR    PUTCR
        RTS
```

MUG'S GAME

Mugsy is a strategy game from Melbourne House, the company responsible for the successful graphic adventure game The Hobbit (see page 540). In this unusual game the player takes the part of Mugsy, a gang leader running a protection racket in the American gangster era of the 1930s.

Mugsy's role is to take various decisions on how many of the gang's 'clients' should be leaned on, how much should be spent on ammunition for the gang, and the amount that should be paid out in bribes to the police. If insufficient money is spent on weapons, the gang will be wiped out. Furthermore, if the police do not receive enough money they raid the gangster's safe and take it all.

The chief character on the screen is Louey, Mugsy's right-hand man. At the beginning of the game Louey appears with a brief outline of the rules. The player makes his strategy decisions in response to prompts from Louey and then waits for the results, during which time one of two animated screens is displayed. The first screen shows a speakeasy in which a gangster fires at a rival. The second shows a street in which a gangster leans out of a car window firing a burst of machine gun bullets.



IAN McKINNELL

Gangland
These are two of the scenes from Mugsy. The first shows part of the question and answer phase of the game, where Louey is informing Mugsy of the current price of 'clients'. The second scene is from one of the animated sequences. Note the white outline around the 'hood' on the stairs. This is an example of attribute masking of character cells

After the break Louey reappears with the results of the previous year's decisions. Provided that adequate funds have been set aside for all of the various payments that have to be made, then the year should end in a profit. The next round then begins, with a repeat of the question and answer routine.

The player's score is given at the end of the game as a percentage. The score depends on how much 'dough' and how many clients have been accumulated as well as the length of time that Mugsy has managed to survive.

Apart from the instruction screens that have a small window at the bottom showing the current number of 'hoods' and 'dough in da safe', etc, the screens consist entirely of graphics pictures. These are drawn on a Spectrum using Melbourne House's graphics package, Melbourne Draw. As a Spectrum high resolution screen will occupy over six Kbytes of memory, there obviously has to be some kind of data compression technique used to cram the code into the free memory available.

An examination of the Mugsy graphics reveals that the programmers have resorted to numerous space-saving techniques. In most cases, pictures are composed from a series of straight lines, and colour is placed very carefully. Using Melbourne Draw commands like FILL and DRAW enables pictures to be built up with the minimum of code — DRAW, for example, requires only two co-ordinates to store a straight line, while bit mapping would need a whole series of points to be plotted to achieve the same effect. The way in which the Spectrum stores information about colour dictates some of the methods that are used — in the animated street scene, a problem arises as a car moves along a road while a face watches from a window. As the Spectrum will not allow more than two colours to be displayed in the same character square, a 'mask' has to be devised to allow colours to be changed very rapidly — otherwise the face will change colour to match the car. Colour attributes (FLASH, BRIGHT, INK and PAPER) are held in a single byte. To produce a foreground mask, the INK attribute, held as the byte's three least significant bits, must be changed. The byte is first ANDed with 248 to set the INK colour to zero, and is then ANDed again with the new INK colour to produce a new mask. The face in fact changes colour to match the car, but then changes back again — but it happens so quickly that the eye is fooled and the face does not appear to change colour at all.

Mugsy's graphics are certainly very impressive, but the game itself palls rapidly. The action is repetitive, and the player soon learns how to juggle the various factors that are needed to stay in business. However, Mugsy does provide aspiring programmers with a good example of how to cram high resolution graphics into a limited space.

Mugsy: For the 48K Spectrum, £6.95
Publishers: Melbourne House, Church Yard, Tring, Herts
Authors: Phillip Mitchell, Greg Cull, Clive Barrett, Russell Comte
Joysticks: Not required
Format: Cassette

DATABASE

Here, courtesy if Zilog Inc, we reproduce a further part of the Z80 programmers' reference card.

Bit Manipulation Group

		REGISTER ADDRESSING							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 53	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE